Final Project Report

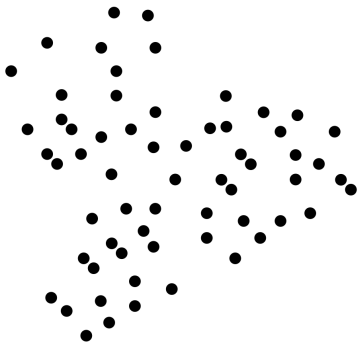
Introduction to Database Systems

Team3: 110062240, 110062323, 110062103, 110062104

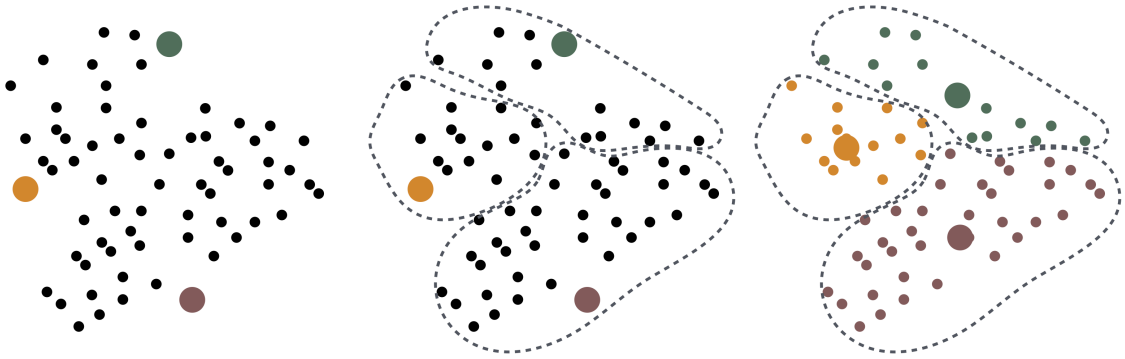
Implementation

Index building Implementation

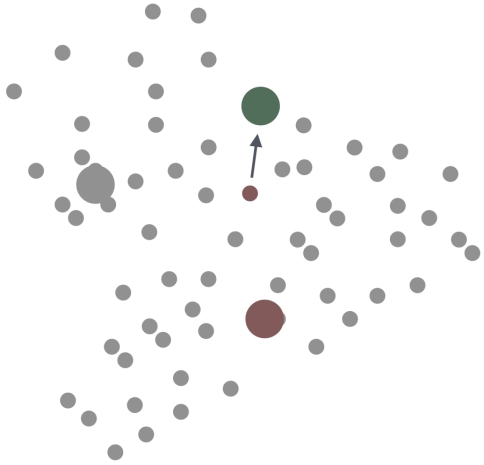
在建立 Index 之實作中，我們使用了 Minibatch K-Means 來處理資料，並加上了各種我們特別實作的技巧，來進一步提升建立 Index 及處理詢問的速度及正確性。



給定的每項資料是長度為 128 的 vector，我們將這些資料表示於二維分佈圖，假設上圖中的每個黑點都為一項資料，我們希望能夠將這些資料分成 K 群，使得每一群中的資料能夠盡量集中。



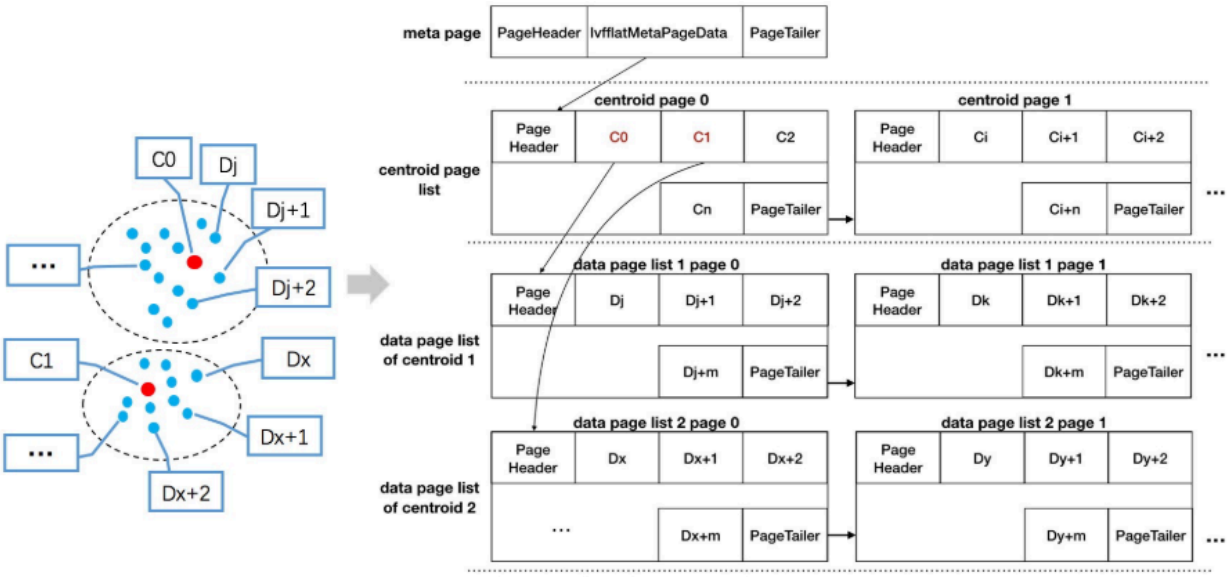
我們首先隨機選擇 K 個點當作未來建立資料群集時的中點，對於其他資料而言，我們會找尋離他最近的中點，擁有同樣一個最近中點的資料將被視為同一個群集。



我們在訓練中逐漸調整這些中點的位置，並重新劃分群集（找尋每個點最近的中點）直到符合我們的收斂條件為止。而對於每個資料而言，在更新中點的位置過後，所隸屬之資料群集之中點不一定是所有中點中最近的（例如上圖中，紅色的資料點離綠色群體之中點較為靠近，因此更新其所隸屬之群集）。因此，我們隨機選擇部分的點，並找尋離這些點最近的中點為何並進行更新。最後，直到所有點離所隸屬的資料群集之中點最近，且沒有任何群集劃分的改變為止。

Index storing Implementation

將資料做劃分後，要將其存入 index table 內，而我們的存法參照了下圖



我們的 index 是特化型的 table，我們直接在該 index 內存 raw data，為的目的是減少讀取 disk 的次數。相比於 VanillaDB 原本的 index 實作，需要透過 index 取得 recordID，再進到對應的 table 內存取資料；我們的實作可以直接在 index 內取得 data，因此少了一次的 IO。且於此同時，我們 index 每個 slot 要存放的資料也變少（因為可以不用存放 recordID 與 record block），schema 細節展現於下圖：

```java
private static Schema schema(SearchKeyType keyType) {
    Schema sch = new Schema();
    for (int i = 0; i < keyType.length(); i++)
        sch.addField(keyFieldName(i), keyType.get(i));
    sch.addField(SCHEMA_RID_BLOCK, BIGINT);
    sch.addField(SCHEMA_RID_ID, INTEGER);
    return sch;
}
```

HashIndex (VanillaDB 本身的實作)

```java
private static Schema schema() {
    Schema sch = new Schema();
    sch.addField(SCHEMA_ID, INTEGER);
    sch.addField(SCHEMA_VECTOR, VECTOR(size:128));
    return sch;
}
```

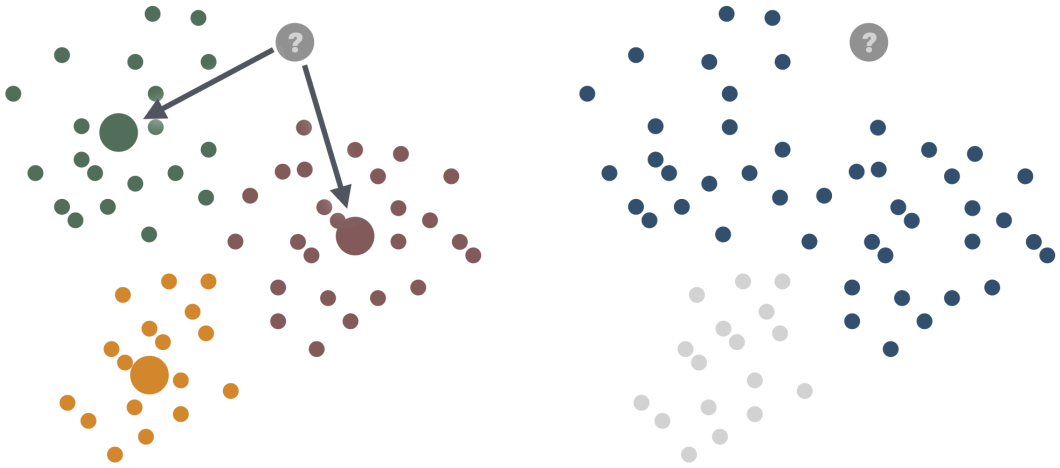
Our index

SIMD Implementation

```java
public class EuclideanFn extends DistanceFn {

    final VectorSpecies<Integer> SPECIES = IntVector.SPECIES_PREFERRED;

    public EuclideanFn(String fld) {
        super(fld);
    }

    @Override
    protected int calculateDistance(VectorConstant vec) {
        // System.out.println("rrrrrrrrrrrrr\nrrrr\nr\nrrrrrrrrrrrrrrrr");
        int[] vec_arr = vec.intVec();

        IntVector tmp = IntVector.zero(SPECIES);

        for (int i=0; i<SPECIES.loopBound(vec.dimension()); i+=SPECIES.length()) {
            IntVector vec_dv = IntVector.fromArray(SPECIES, vec_arr, i);
            IntVector query_dv = IntVector.fromArray(SPECIES, int_query, i);
            IntVector diff = vec_dv.sub(query_dv);
            tmp = diff.mul(diff).add(tmp);
        }

        int sum = tmp.reduceLanes(VectorOperators.ADD);
        return sum;
    }

}
```

我們除了實作的Java.incubator.vector的SIMD外，也將原本浮點數的運算換成整數運算，進而優化了計算距離所需的時間。然而，在測試過後發現 SIMD 的表現不如預期，效果與未用 SIMD 相當甚至更差，因此我們在提交的程式碼中並未採用。

Search Improvements

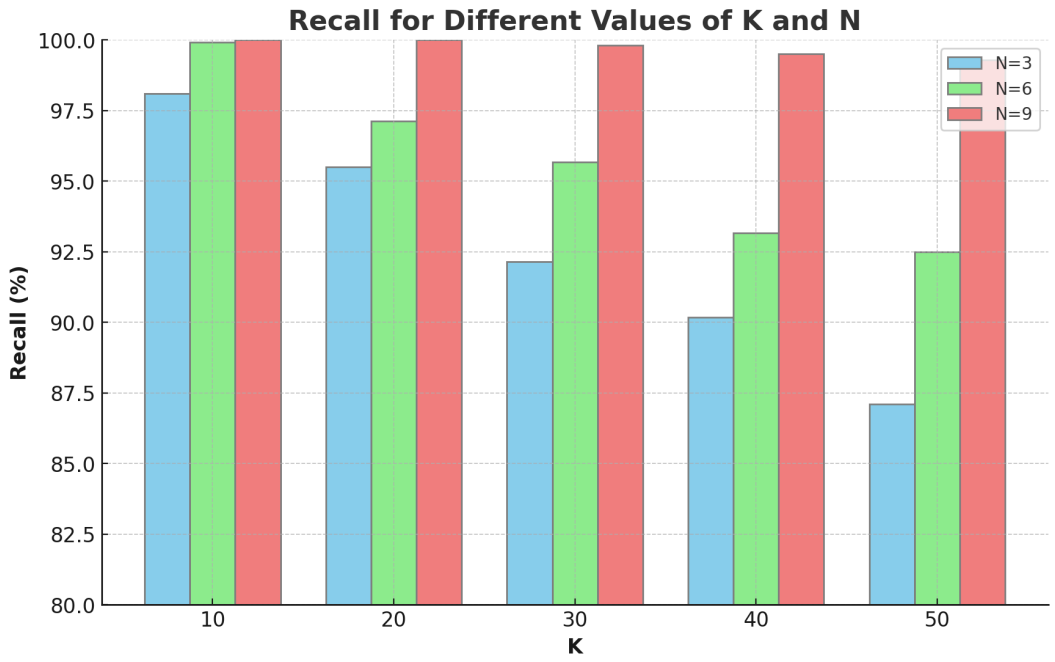


在進行詢問時，我們必須找尋離詢問數值最近的 20 個資料點，首先，我們會找尋離詢問資料點最近的 N 個群集中點，而選中的群集中的點便是我們會去進一步篩選的資料集。我們會依序掃過這些點，並維護一個 max heap，依序將這些點與詢問點的距離放進 heap 中，只要 heap 的資料數量大於 20 便將頂端的資料拿出，確保 heap 中的資料一定是前 20 小的距離。這樣一來，既能大幅度減少搜尋的時間（維護 heap 的時間為常數時間，只需要掃過 N 個群集中的所有點），也能提高詢問的正確性。

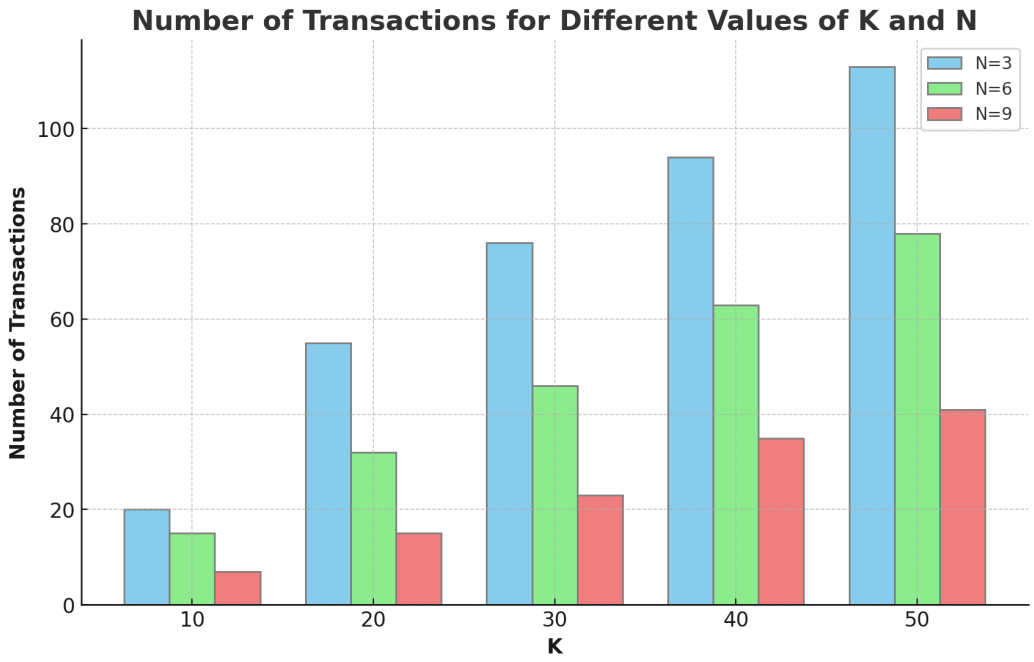
Experiments

環境配置：**12th Gen Intel(R) Core(TM) i7-12700H / Ubuntu 20.04 / RAM 16G**

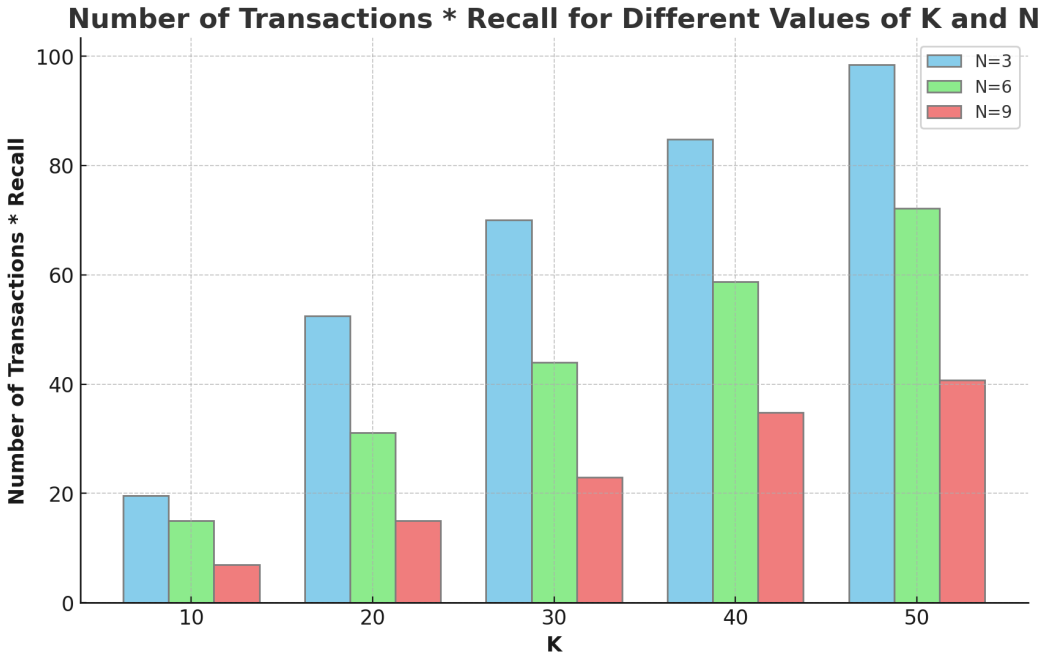
由於以實際測試的數據來進行實驗會耗費較多的時間計算 recall，因此，我們採用了較小的數據來衡量我們的實驗情形：我們以 Number of Items = 200000 及 Benchmark Interval = 1000 作為基準來進行測試，K 為我們劃分的資料群集數量，N 為我們選中的最近群集數，並將這些群集中的資料依序放進 max heap 中選擇前 20 小者。



上圖為不同的 K 及 N 值之 recall 的比較，可以觀察到只要搜尋的資料群集數 N 夠大，由於看過較多的資料，因此 recall 的表現都不會太差，然而，若是 K 值變大而 N 值並未隨之變大時（例如 N=3, K=50 及 N=3, K=10），由於劃分的群集數量 K 變多，使得固定看同樣數量的資料群集時能夠看過的資料數目變少，而導致 recall 會隨之下降。



上圖為不同的 K 及 N 值之 1 秒內能完成之 transactions 的比較，可以觀察到固定 K 值且 N 值變大時，由於搜尋的資料數量變多了，latency 也會隨之提高。然而當 K 變多時，同樣的 N 值由於因為同個資料群集中的資料數量變少，因此能夠完成的 transactions 數量也會隨之提升。



上圖為將能夠完成的 transactions 數量及 recall 之相乘之值（比較 performance），由於 recall 數量都不太低，因此圖形分佈情形與能夠完成的 transactions 數量較為類似。然而，我們也可以藉此觀察到，在我們的設計中，一旦將 K 值提升越大，便有較多得以更為精細調整 N 值的空間，來去決定我們要看過的資料數量（因為單一資料群集的大小變小了）。由於資料看得愈多，recall 值就會隨之上升，然而 latency 也會隨之下降，因此，只要能夠更為精細的決定要看過的資料數量（將 K 值提升），我們便能更好地去找到最適合的 N 值，來取得最好的 recall 及 throughput 的平衡。而在最後以 Number of Items = 900000 的版本中，由於建立 Index 的時間限制，我們採用了 K=15 及 N=1 的選擇，以在 performance 中取得最佳的平衡。

Reference

https://dl.acm.org/doi/pdf/10.1145/3318464.3386131